# SystemCによるビヘイビア合成入門

## 第8回/最終回 ビヘイビア設計の流れと今後の動向

桜井 至

SystemC言語を入力とするビヘイビア合成技術の基礎について解説した連載の最終回をお届けする．今回は，アルゴリズム記述の作成からRTL記述の生成までの具体的な作業手順と各工程の注意点について説明する．生成されるRTL記述の性能にこだわる場合，ハードウェアの構成を意識しながらアルゴリズム記述に手を加えていく必要がある．（編集部）

本連載も今回で最終回を迎えます．この講座では，ビヘイビア合成の基本的な機能や設計フロー，記述方法などについて紹介してきました．現在，実際の設計でビヘイビア合成ツールを利用している読者の方はまだ少ないと思いますが，ビヘイビア合成とは何かを知りたい方や，今後の導入を検討されている方の参考になったのであれば幸いです．

最終回になる今回は，アルゴリズム記述からハードウェア化を考慮したビヘイビア記述への変更の過程，今後の動向，OSCI（Open SystemC Initiative；SystemCの標準化・普及推進団体）の標準化作業などについて解説します．

### ● ビヘイビア・レベルの設計フローをおさらい

最初に，アルゴリズム記述を作成して実際にビヘイビア合成を実行するまでの設計フローについて説明します（**図1**）．

まず，アルゴリズム記述について，どの部分をハードウェア化するかの検討を行います．アルゴリズム記述の中にはハードウェア化に必要のない部分やソフトウェアとして実現する部分が含まれているので，それらを分離します．ハードウェア化する部分が決まったら，ハードウェア化を考慮した分割や記述の変更を行います．その際に，TLM（transaction level modeling）モデルによる協調検証環境を用いれば，システム検証やハードウェアとソフトウェアのトレードオフ解析が行いやすくなります．

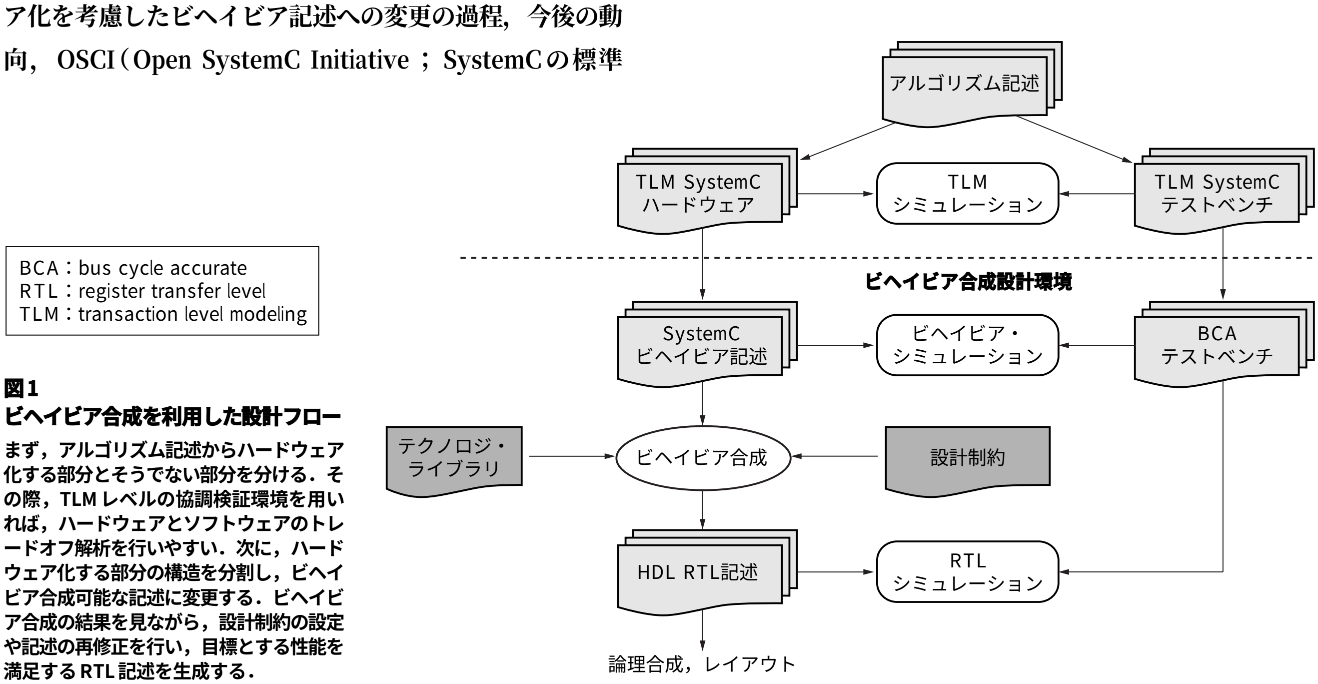



**図1**
**ビヘイビア合成を利用した設計フロー**
まず，アルゴリズム記述からハードウェア化する部分とそうでない部分を分ける．その際，TLMレベルの協調検証環境を用いれば，ハードウェアとソフトウェアのトレードオフ解析を行いやすい．次に，ハードウェア化する部分の構造を分割し，ビヘイビア合成可能な記述に変更する．ビヘイビア合成の結果を見ながら，設計制約の設定や記述の再修正を行い，目標とする性能を満足するRTL記述を生成する．

次に，ハードウェア化する部分の構造を分割します．分割にはモジュールやプロセスによる分割があります．ビヘイビア合成では，このような分割が最終的に得られるRTL回路の性能を決めるので重要な作業であると言えます．さらに，ビヘイビア合成できない記述をビヘイビア合成可能な記述に変更し，必要に応じてデータ・タイプのビット幅の指定を行います．

この時点で，ほぼビヘイビア合成が可能となりますが，合成の結果を見ながら，ビット幅の再指定や記述の変更，ループや配列の展開などの指示を行い，最終的に目標性能を満足するRTL（register transfer level）回路を得ます（p.143のコラム「ビヘイビア合成ツールの実例」を参照）．

なお，SystemCを用いてRTLと同等な記述レベルで設計することも可能ですが，この場合，従来どおりHDLで記述したほうが効率的で，検証も高速です．SystemCを使用するメリットはビヘイビア・レベルやTLMレベルで設計できることにあります．

以下では各工程について，順番に説明していきます．

### ● 性能重視の場合はハードウェア化を考慮して記述

ビヘイビア合成は，RTL記述より抽象度の高い記述を用いるため，その入力であるビヘイビア記述についてはハードウェア構造（アーキテクチャ）を考慮する必要はないと思われるかもしれません．C/C++を用いたアルゴリズム記述をそのままビヘイビア合成できる場合もありますが，目標とするハードウェアの性能を満足するRTL記述が生成される保証はありません．

必要な性能を備えるRTL記述を確実に得るためには，ある程度ハードウェア構造を考慮する必要があります．ハードウェア構造を考慮していないアルゴリズム記述を入力すると，期待はずれのレイテンシや面積を持つRTL記述が生成される可能性があります．そのため，一般には純粋なアルゴリズム記述を用いずに，ハードウェアを考慮したビヘイビア記述に書き換えます．

一方，従来のHDLによるRTL設計では，RTL設計を開始する時点で詳細なハードウェア構造を決める必要があります．レジスタの位置や必要な演算リソース，マルチプレクサ，動作を制御するステート・マシンなど，具体的なハードウェア構造に基づいてRTL記述を作成していきます．ビヘイビア合成を用いればこのような詳細な構造はツールが自動的に生成しますが，より上位の構造は設計者が考慮しなければなりません．

### ● 必要最低限のデータを扱うように書き換える

問題の理解を容易にするために，簡単なアルゴリズム記述の構造例を**リスト1**に示します．アルゴリズムを構成する各動作は関数として定義されています．関数の引き数にはデータ配列が渡され，計算された結果がデータ配列に戻ります．**リスト1**では，ライン長として4096の1次元配列を定義していますが，この配列サイズは，画像処理ではフレーム，画像ブロック，ライン長などになると思います．

アルゴリズム記述では，フローを組み立てやすいように引き数に配列をとり，それぞれが独立に動作する関数を作成することがあります．しかし，そのような関数で構成されたアルゴリズム記述をビヘイビア合成すると，配列はメモリにマッピングされます．その場合，それぞれの関数の間にメモリ・アクセスが入るため，処理に多くのサイクル数がかかることになります．また，配列をレジスタに展開するとレジスタ数が多くなるため，面積が増えてしまいます．いずれにしても，良いRTL記述が生成されません．

**リスト1　簡単なアルゴリズム記述の構造例**

```
#define MAX 4096

void algorism()
{

  int data[MAX];

  . . . . .

  // メイン・アルゴリズム
  // 関数A
  void func_A(int *data, ..) {
    int a;
    for(int i=0; i<MAX; i++) {
       a = *(data + i);
       // アルゴリズムA
       . . .
       *(data + i) = . . . ;
    }
  }

  // 関数B
  void func_B(int *data, ..) {
    int b;
    for(int i=0; i<MAX; i++) {
       b = *(data + i);
       // アルゴリズムB
       . . .
       *(data + i) = . . . ;
    }
  }

  // 関数B
  void func_C(int *data, ..) {
    int c;
    for(int i=0; i<MAX; i++) {
       c = *(data + i);
       // アルゴリズムC
       . . .
       *(data + i) = . . . ;
    }
  }

  . . .

}
```

ハードウェア化を考えた場合，冗長なデータを扱うことは，その分だけハードウェア・コストの増加を招きます．そのため，計算に必要な最低限のデータだけを扱うようにします．例えば，**リスト1**の記述から，全体を制御するforループをいちばん外側に出し，その中に，必要なデータだけを用いて計算する関数を入れるようにします．計算に必要なデータが5個であれば，例えば**リスト2**のようにシフト・レジスタの構造を定義します．

もし，実際のハードウェアにおいて，データ入力の開始を示す信号と終了を示す信号で制御されるのであれば，このforループも必要ありません．一度に五つのデータを入力するのであれば，シフト・レジスタも必要ありません．このような場合，forループは外部システムまたはテストベンチで定義されることになります．

アルゴリズム記述では，ハードウェア化の対象となる記述と，それ以外のテストベンチ側に属する記述が混在している場合がよくあります．まずは，ハードウェア化する対象の記述を分離し，さらに，ハードウェア化に必要なインターフェースの仕様を明確にします．

SystemCのTLMによるハードウェアとソフトウェアの協調検証環境を用いることで，インターフェース仕様の検討やハードウェアとソフトウェアの分割のトレードオフ解析が容易になります（詳しくはSystemC関連の参考文献を参照していただきたい）．

## ● ハードウェアを分割・階層化

アルゴリズム全体を一括してビヘイビア合成できたとしても，やはり生成されたRTL回路の性能が上がらない場合があります．また，生成したRTL回路のサイズが大き過ぎて，論理合成のときに処理できないこともあります．大規模なビヘイビア設計では，ハードウェア構造を考慮して分割する必要があります．

SystemCでは，SC_MODULE構文を用いてモジュールを作成し，階層構造を構築します．モジュールには，入出力ポートや内部信号，メンバ変数，メンバ関数などを定義し，さらにSC_CTHREADプロセスやSC_METHODプロセスを作成して，必要な構造を定義します（**図2**）．また，プロセスには，ポートや信号値の読み出し，書き込みを行う方法を示したプロトコルを定義します．モジュールやプロセスの間の接続には，信号としてsc_signal<>を用います．

階層分割の際に考慮すべき要素には，面積（RTL回路のサイズ），並列性，機能，処理サイクル数などがあります．サイズは，ビヘイビア合成や論理合成で扱える大きさに抑えることが必要です．ビヘイビア合成では，サイズを大きくするとリソースを共有化する可能性が高くなるというメリットがあります．モジュールを小さく分割すると，リソースの共有化はモジュールの中でしか行われないことになります．つまり，共有化が十分に行われない場合があります．

ビヘイビア合成によって生成されるハードウェアは，ビヘイビア記述どおりに順序的に実行されます．一つの処理に10サイクルが必要な場合，二つの処理を順序的に実行すればトータルで20サイクルが必要です．しかし，それぞれの処理で入出力が独立しており，並列的に実行できるのであれば，並列化することにより10サイクルで処理できます．一般に面積とサイクル数はトレードオフの関係にあるので，階層分割は適切に行わなくてはなりません．

また，メモリ・アクセスなどの制御部とパイプラインなどのデータ処理部を一つの記述に混在させると，制御部がサイクルのボトルネックとなり，性能が上がらない場合が

**リスト2　計算に5個のデータが必要な場合の変更例**

```
#define MAX 4096

void algorism()
{

  int data[MAX];
  int din[5];
  int tmp[5];
  . . . . .


  // メイン・アルゴリズム
  for(int i=0; i<MAX; i++) {

    // シフト・レジスタの定義
    for(int j=0; j<4; j++) {
      din[j] = din[j+1];
    }
    din[4] = data[i];

    if (i > 4) {
      // 関数A
      void func_A(int *din, ..) {
         // アルゴリズムA
         . . .

      }

      // 関数B
      void func_B(int *tmp, ..) {
         // アルゴリズムB
         . . .
      }

      // 関数C
      void = func_C(int *tmp, ..) {
         // アルゴリズムC
         . . .

      }

    }  // if文の終了
  } // forループの終了
  . . .

}
```

あります．さらに，ビヘイビア合成ツールによっては制御部とデータパス部の記述方法やビヘイビア合成の手法が異なるケースもあります．そのようなときにも機能面での分割が必要です(**図3**)．

アルゴリズムの分割には複数の要素が関与するので，設計者は目標とするハードウェアをよく理解していることが必要です．ゲート・レベル設計の時代は，設計者の腕の見せ所は，どのように論理ゲートを接続してゲート・レベル回路を構築するかでした．RTL設計では，レジスタやリソースの配置，ステート・マシンの定義といったRTL回路の構造を決めることが腕の見せ所になりました．ビヘイビア・レベル設計ではさらに抽象度が上がり，優れたハードウェアを実現するために，アルゴリズムをいかに適切に分割・構築するかという点が重要になります．今後，ビヘイビア合成がより進化して高性能になったとしても，優れたハードウェアを実現するためには，ハードウェア設計の知識や経験が必要です．

## ● ビヘイビア合成可能な構文に書き換える

ハードウェア化する部分について，ビヘイビア合成がサポートしていない構文や記述があれば，それをビヘイビア合成可能な記述に変更する必要があります．現在，ポインタの一部はビヘイビア合成ツールによってサポートされていますが，これは関数の中で配列を参照し，結果を戻す場合に限定されます．ポインタを用いた以下のような動的な配列宣言は，一般にサポートされていません．

```
int *data;
data = new int;
```

ハードウェア化する場合，配列については配列サイズを明確に指定して宣言しなければなりません．配列のサイズは，目標とするハードウェア構造から容易に確定できる場合もありますし，アルゴリズム記述だけでは確定が難しい場合もあると思います．不必要に大きな配列サイズを指定すると，ハードウェア・コストの増大を招きます．なお，ビヘイビア合成でサポートされている構文や記述スタイルは，OSCIのビヘイビア合成サブワーキング・グループで標準化されています[注1]．

## ● データ・タイプを変更する

アルゴリズム記述では，int，short，longなどの

注1：SystemC入力のビヘイビア合成ツールは，基本的にOSCIの合成サブセットに準じているが，詳細なサポート構文はツールごとに異なる場合がある．詳細は各ツールの開発元に問い合わせていただきたい．

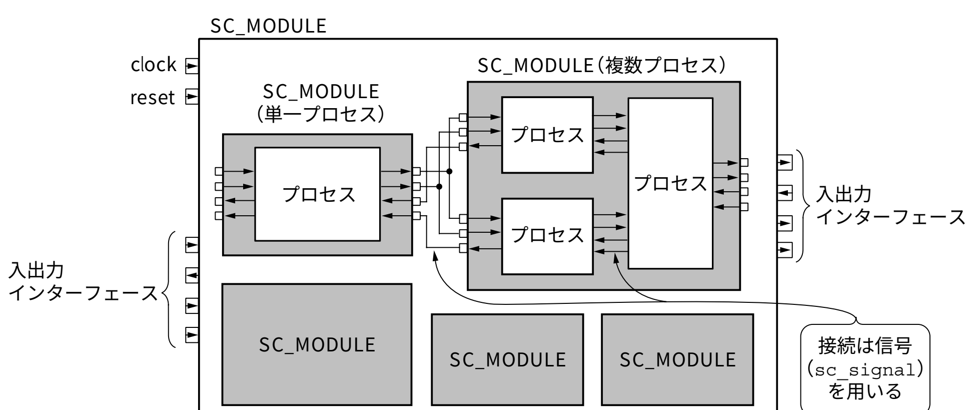



**図2**
**アルゴリズムのモジュールとプロセスへの分割**
ハードウェア化を考慮する場合，アルゴリズムをSC_MODULE構文によってモジュールに分割し，階層構造を構築する．モジュール内部ではSC_CTHREADプロセスやSC_METHODプロセスを作成し，必要があればさらに詳細な構造を定義する．

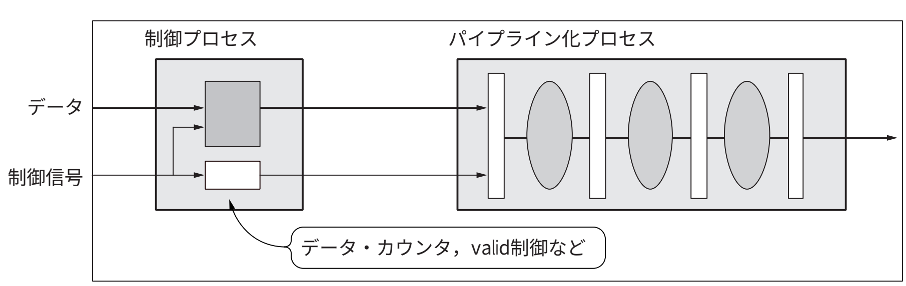



**図3**
**制御部とデータパス部の機能分割例**
制御部とパイプラインなどのデータパス部を一つの記述に混在させると，制御部が処理サイクルのボトルネックとなり，性能が上がらない場合がある．さらに，制御部とデータパス部のビヘイビア記述方法や合成手法が異なる場合もある．そのような際には，機能を考えて制御部とデータパス部に分割する．

標準的なC/C++のデータ・タイプを使用します．これらのデータ・タイプに対して，ビヘイビア合成ではデフォルトのビット幅を割り付けます．例えば，charは8ビット，shortは16ビット，intは32ビットなどとなります．**表1**にデフォルトのビット幅の割り付け例を示します．

しかし，このようなC/C++の標準データ・タイプを用いた場合，ビヘイビア合成によって必要以上のビット幅を持つ演算リソースが割り付けられてしまう可能性があります．例えば，24ビットの演算で計算できるアルゴリズムに対してintを用いたことで32ビットの演算リソースが割り付けられ，その結果，面積が増加してしまうといった状況です．ビヘイビア記述ではC/C++のデータ・タイプはできるだけ使用せず，sc_int<>やsc_uint<>を使用してビット幅を明確に指示したほうが無難です．

また，計算式でビット幅が異なる場合，左辺の代入先のビット幅に合わせるように右辺の演算のビット幅が確定されます．例えば，以下のような式の場合，左辺が12ビットであるため，12ビットの加算器が割り付けられます．

```
sc_uint<12> x;
sc_uint<16> a, b, c;
x = a + b;
```

しかし，右辺の式が複雑な場合には，すべての演算リソースのビット幅を最適化できないことになります．その場合，以下のようにキャストを用いてビット幅を指定します．

```
x = a + (sc_uint<16>)( b * c ) ?
                    (sc_uint<12>)( e * f )
```

ビット幅の指定は，生成されるハードウェアの面積に大きな影響を与えることがあるので，適切な指定が必要です．さらに，doubleやfloatなどの浮動小数点データ・タイプについても，どのように実現するかの検討が必要です．sc_fixed()を用いた固定小数点演算は，整数演算より演算リソースが大きくなるので，できるだけ整数演算で処理できるように置き換えます．また，符号付き演算は符号なし演算より演算リソースが大きくなる傾向があります．符号なしで済むのであれば符号なしを指定します．

このように，アルゴリズム記述の段階ではあまり気にしないところでも，ハードウェア化する場合には明確化したほうがよい部分があるので注意が必要です．

**表1　C++データ・タイプに対するSystemCデータ・タイプのサイズ**

| C++データ・タイプ | ビヘイビア合成におけるサイズの割り付け |
|---|---|
| bool | sc_uint<1> |
| char | sc_int<8> |
| unsigned char | sc_uint<8> |
| short | sc_int<16> |
| unsigned short | sc_uint<16> |
| int | sc_int<32> |
| unsigned int | sc_uint<32> |
| long | sc_int<32>　または sc_int<64> |
| unsigned long | sc_uint<32>　または sc_uint<64> |
| long long | sc_int<64> |
| unsigned long long | sc_uint<64> |

### ● ハード技術者向きか，ソフト技術者向きか

ビヘイビア記述の設計資産は，ビヘイビア合成によってクロック周期やテクノロジ（プロセス技術やライブラリ）を変えてスケジューリング/アロケーションし直すことができます．そのため，RTL記述の設計資産より汎用性が高くなります．また，検証に必要なシミュレーション時間の短縮も期待できます．

SystemCはC++のクラス・ライブラリを用いて実現しているため，C++のオブジェクト指向に基づいた設計資産の再利用も可能です．クラス・ライブラリとして設計資産を構築し，それらを再利用していくことになります．また，記述スタイルは異なるとしてもソフトウェア設計者と同じ言語を使用することになるので，ソフトウェア設計者と協力しやすいのではないでしょうか．

「ビヘイビア記述を習得するには，C++の知識は持つがハードウェアを知らないソフトウェア設計者が良いのか，RTL設計の知識は持つがC++/SystemCを知らないハードウェア設計が良いのか」，という質問を受けることがあります．そのような場合，「ビヘイビア合成の目標としているのはハードウェア設計なので，ハードウェア設計者のほうでしょう」と筆者は答えています．HDL設計の知識を持つ設計者がC++/SystemCといった言語を勉強することはなんとかなりそうですが，ソフトウェア設計者がハードウェア設計そのものを理解することはたいへんです．設計の目標がハードウェアである限り，ハードウェアの実現方法の知識はビヘイビア記述であっても絶対に必要だからです．

HDL設計者からすると，SystemC言語は複雑で難解なように感じられると思います．しかし，ビヘイビア記述で用いるC++/SystemC言語はそれほど複雑ではありません

(C++のすべての機能を理解している必要はない).HDL記述に慣れている設計者であれば,SystemCへ移行することは比較的容易だと思います.

## ● チャネル合成や等価性検証などの環境整備に期待

SystemCやビヘイビア合成が,今後,どのように使用されていくのかは,まだ不明確な部分がありますが,徐々に普及していくと筆者は信じています.今後は,次のようなビヘイビア合成に関連する技術が提供されるものと考えられます.

### 1) チャネル(インターフェース)合成

現状のビヘイビア合成では,入出力インターフェースにバス・サイクル精度(BCA : bus cycle accurate)レベルの記述が必要とされています.バス・サイクル精度レベルでは,RTLと同じようにポートとビット幅を定義し,入出力動作もクロック精度で定義します.チャネル合成は,SystemCのチャネル・モデルから自動的に詳細なインターフェースの信号やプロトコル動作を行うハードウェアを生成します.そのため,詳細なインターフェース仕様を知らなくても,`read()`,`write()`などのアクセス関数を使用するだけでインターフェース仕様に基づいたプロトコル部を自動的に生成することができます.

SystemCを用いるメリットの一つはチャネル・モデルにあります.チャネル合成が実用化されれば,SystemCはさらに使いやすくなることでしょう.

### 2) 等価性検証(equivalence checking)

RTL設計では,RTLとゲート・レベルで機能が一致しているかどうかを,等価性検証ツール(一種のフォーマル・ベリフィケーション・ツール)を用いて確認しています.等価性検証を利用すると,テストベンチを用意しなくても,アルゴリズム的な機能の等価性を確認できます.シミュレーションを行う必要がないため,短時間に機能を検証できるわけです.

現在,ビヘイビア・レベルとRTLの機能の等価性は,シミュレーションで確認しています.ビヘイビア・レベル・シミュレーションの時間は比較的短いのですが,RTLシミュレーションには長い時間が必要になる場合があります.現在,RTLとゲート・レベルの等価性検証が使われているように,いずれビヘイビア・レベルとRTLの間の等価性検証も必要とされるようになることでしょう.

### 3) グラフィカルな設計環境

SystemCを用いた設計を行う場合,モジュールの分割や構築の作業はグラフィカルな画面(例えば,ブロック図エディタ)上で行えばわかりやすくなります.また,HDL設計に設計スタイル・チェッカがあるように,ビヘイビア合成にも記述スタイルのチェッカやデバッグ環境があれば便利です.現状,このような環境は十分整備されているとは言えませんが,SystemCが普及すればより良い環境が提供されてくることでしょう.

このほかにも,消費電力を考慮したビヘイビア合成のスケジューリングや最適化の実行,ハードウェアとソフトウェアの協調設計など,今後も多くの技術が開発されると考えられます.それによって,さらなる設計の効率化が図られていくことでしょう.

## ● 言語,合成,モデリング,検証について標準化

最後にOSCIにおける標準化活動について紹介します.OSCIでは,SystemC言語の標準化や啓蒙を中心に活動し,SystemCのリファレンス・シミュレータの開発と提供を行っています.SystemCの構造を**図4**に示します.SystemCの基本構文やデータ・タイプはC++の上に構築されています.OSCIのホームページ(http://www.systemc.org/)からSystemCライブラリ,カーネル,ヘッダ・ファイルなどのパッケージをダウンロードすることができます.それらを使うことによって,SystemCで記述した設計データをシミュレーションしたり,標準的なC++コンパイラを用いてコンパイルできるようになります.本連載で紹介した記述例も,そのままシミュレーションすることができます.HDLシミュレータは,現在でこそフリーのものを使えるようになりましたが,登場初期のころは高価でした.SystemCの検証環境はほぼ無償で構築することができるので,導入しやすいと言えます.

SystemCは,2000年にバージョン1.0が,2002年にバージョン2.0がリリースされました.現在のSystemCの最新バージョンは2.1となります.SystemCのバージョンとサポート範囲の変遷を**図5**に示します.SystemCは,バージョン3でリアルタイムOSモデルを,バージョン4でミックスト・シグナル・モデルをサポートすることが予定されています.

OSCIには,現在,以下のワーキング・グループが設置されており,実際にSystemCを使用するうえでのモデリン

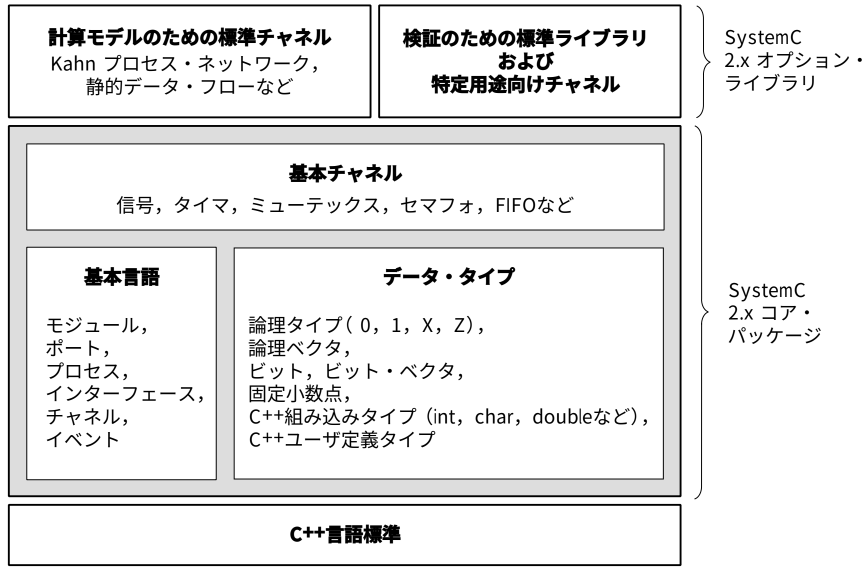



**図4**
**SystemC言語の構造**
**SystemCの基本構文やデータ・タイプはC++の上に構築されている．そのため，SystemCで記述された設計データはそのままシミュレーションしたり，標準的なC++コンパイラを用いてコンパイルできる．**

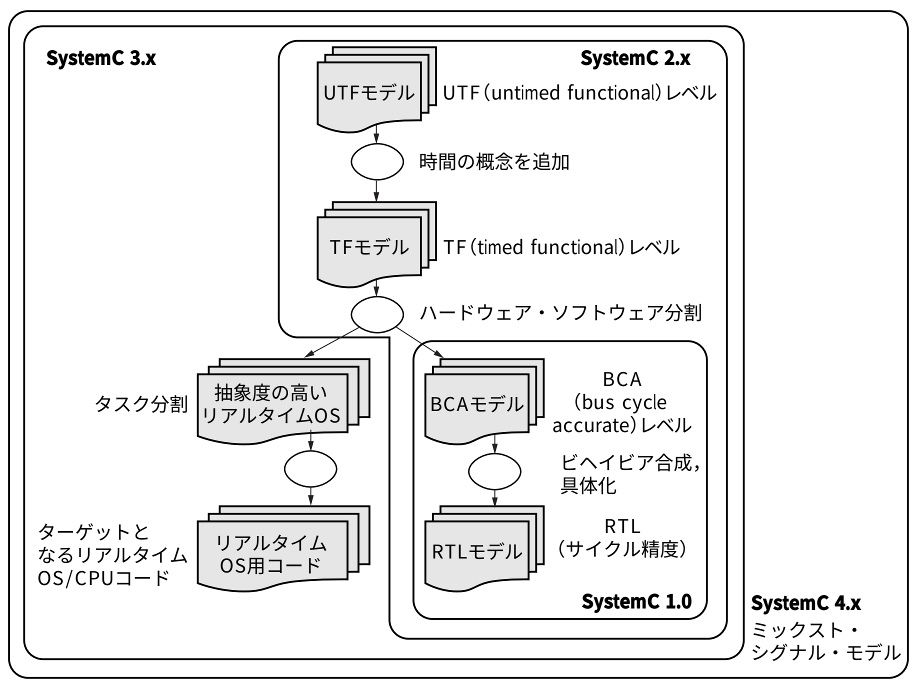



**図5**
**SystemCのバージョンとその機能**
**SystemCは，2000年にバージョン1.0が，2002年にバージョン2.0がリリースされた．現在のSystemCの最新バージョンは2.1．また，バージョン3でリアルタイムOSモデルが，バージョン4でミックスト・シグナル・モデルがサポートされる予定．**

グの標準化作業を進めています．

- **言語標準化**
- **TLM**
- **ビヘイビア合成**
- **検証ライブラリ**

本連載で紹介した記述例は，ほぼビヘイビア合成サブワーキング・グループが作成したビヘイビア合成サブセットの仕様に基づいています．

本連載も今回で最終回となりました．今後，SystemCを用いた高位設計が普及することはまちがいないと筆者は考えています．長い間どうもありがとうございました．

**参考・引用*文献**


(1) OSCI（Open SystemC Initiative），SystemC User's Guide / Functional Specification，2004.
(2) 長谷川裕恭，小川丈博，連載・SystemCを使ったシステム設計，Design Wave Magazine，2003年7月号～2004年7月号.
(3) Thorsten Grotker，Stan Liao，Grant Martin，Stuart Swan（柿本勝，河原林政道，長谷川隆 監訳）；SystemCによるシステム設計，丸善，2003年.
(4) 大村正之，深山正幸；C/C++によるVLSI設計，共立出版，2003年.
(5) OSCI，SystemC Synthesizable Subset，2005.

## COLUMN ビヘイビア合成ツールの実例

ビヘイビア合成ツールはいくつかのEDAベンダが出荷しています．ここでは本連載を執筆するにあたって筆者が使用した米国Forte Design Systems社の「Cynthesizer」を例に，実際のビヘイビア合成ツールのサポート機能を紹介します．

Forte社では，2004年4月にCynthesizerを正式発表しました．本ツールは，SystemCを用いたビヘイビア記述に対してスケジューリング，演算リソースの割り付け，および共有化を行い，論理合成可能なRTL記述を出力します(**図A-1**)．

また，同社は，本ツールと組み合わせて高位設計環境を構築するBDW(Behavioral Design Workbench)と呼ぶソフトウェアも提供しています．BDWは，テクノロジ・ライブラリや設計ファイルなどの環境指定，ビヘイビア合成やシミュレーションの実行制御を行います．SystemCシミュレータとしては，OSCIが公開しているSystemCシミュレータや米国CoWare社のSystemCシミュレータを使用できます．ビヘイビア合成されたRTL記述の検証のため，SystemCシミュレータとVerilog HDLシミュレータの協調シミュレーションを行うインターフェースも用意しています．

ビヘイビア合成に必要なデータは，演算子に対応したリソース・ライブラリ，ハードウェア化の対象となるビヘイビア記述，設計制約ファイルなどです．リソース・ライブラリは，入出力ビット幅ごとに異なる加減算器，乗算器，論理演算，関係演算，論理演算などのコンポーネントで構成されます．これらのライブラリは，論理合成用のテクノロジ・ライブラリから論理ゲート回路にマッピングされ，その結果をもとにタイミングと面積の情報が抽出されます．同一演算に対して，高速用，低速用など，複数のタイプの演算リソースを持ちます．

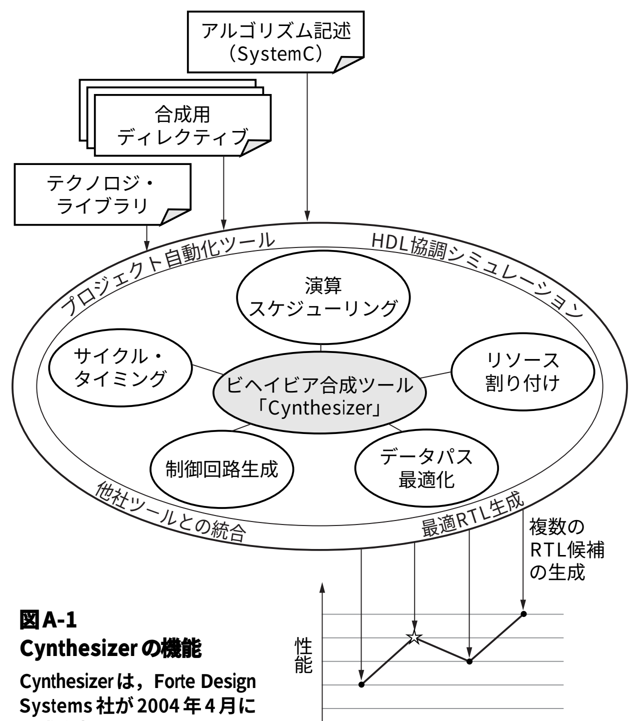
ビヘイビア合成のスケジューリングは，リソース・ライブラリとクロック周期に依存します．例えば，加算器のタイミングが5ns，乗算器のタイミングが10nsとすると，クロック周期が20nsであれば，1サイクルで二つの演算を実行できます．クロック周期が10nsであれば，2サイクルで演算を実行します．クロック周期が5nsであれば，パイプライン乗算器を用いて3サイクルで演算を実行します．論理合成の際にタイミング・エラーが生じないように，演算リソースに論理ゲート・レベルの正確なタイミング情報が必要です．

ビヘイビア合成の対象となる設計データは，C++/SystemCで記述します．ビヘイビア合成可能なC++構文には，演算子，条件文，配列，関数だけでなく，テンプレートや継承クラスなども含まれます．そのため，設計の再利用が容易になるオブジェクト指向設計を採用できます(これらの構文については，本連載の中で紹介した)．

ビヘイビア記述に対して，クロック周期やレイテンシなどの設計制約条件を与え，スケジューリングやリソースの割り付け，共有化を実行します．必要があれば，記述内のループの展開や配列の展開，パイプライン化などを指定します．何度も使用される数式に対して，データパス最適化ツールを用いて，複数の演算子をまとめた固有のコンポーネントを作成することもできます．記述固有のコンポーネントを作成することで，生成されるRTL回路の性能を改善できます．

生成されたRTL記述については，テストベンチを用いてビヘイビア記述との等価性を確認します．SystemCを用いると，同じテストベンチを用いてRTL記述の検証を行えます．ビヘイビア合成ではレイテンシが変わるため，外部との同期をとるためにハンドシェイクが必要です．ハンドシェイクが不完全な場合，レイテンシの変化に対応できず，結果が一致しない場合があります．ビヘイビア・レベルとRTLの間の等価性検証ツールは，まだ開発段階であるため(スタートアップ企業などからようやく製品発表が始まった段階)，現状では，シミュレーションで結果の一致を確認せざるをえません．

ビヘイビア合成されたRTLファイルは，BDW環境からサード・パーティ製の論理合成ツールへ自動的に送られます．また，サード・パーティ製のRTL構造解析ツールや消費電力解析ツールなどへ渡すこともできます．




**図A-1**
**Cynthesizerの機能**
Cynthesizerは，Forte Design Systems社が2004年4月に正式発表した．

**さくらい・たかし**
**フォルテ・デザイン・システムズ(株)**

**<筆者プロフィール>**
**桜井 至．EDA業界に入り，Verilog HDLから始まり，VHDL，Verilog-AMSへと進み，現在はSystemCを担当．さらに合成ツールとしては，論理合成，アナログ合成と進み，現在はSystemCとビヘイビア合成に従事．HDLと合成の大好き人間と言えるかもしれません．著書に「HDL設計入門」，「HDLによるディジタル設計の基礎」，「LSI設計の基礎技術」(テクノプレス社)，および「Verilog-AMS入門」(CQ出版社)がある．現在，フォルテ・デザイン・システムズ(http://www.forteds.com/)に所属(フォルテでは人材募集中)．**